

# Efficient Truss Maintenance in Evolving Networks

Rui Zhou  
Center for Applied Informatics  
College of Engineering &  
Science  
Victoria University  
Melbourne, Australia  
Rui.Zhou@vu.edu.au

Chengfei Liu  
Faculty of Information and  
Communication Technologies  
Swinburne University of  
Technology  
Melbourne, Australia  
cliu@swin.edu.au

Jeffrey Xu Yu  
Department of Systems  
Engineering & Engineering  
Management  
The Chinese University of  
Hong Kong  
Hong Kong, China  
yu@se.cuhk.edu.hk

Weifa Liang  
Research School of Computer  
Science  
Australian National University  
Canberra, Australia  
wliang@cs.anu.edu.au

Yanchun Zhang  
Centre for Applied Informatics  
College of Engineering &  
Science  
Victoria University  
Melbourne, Australia  
Yanchun.Zhang@vu.edu.au

## ABSTRACT

Truss was proposed to study social network data represented by graphs. A  $k$ -truss of a graph is a cohesive subgraph, in which each edge is contained in at least  $k - 2$  triangles within the subgraph. While truss has been demonstrated as superior to model the close relationship in social networks and efficient algorithms for finding trusses have been extensively studied, very little attention has been paid to truss maintenance. However, most social networks are evolving networks. It may be infeasible to recompute trusses from scratch from time to time in order to find the up-to-date  $k$ -trusses in the evolving networks. In this paper, we discuss how to maintain trusses in a graph with dynamic updates. We first discuss a set of properties on maintaining trusses, then propose algorithms on maintaining trusses on edge deletions and insertions, finally, we discuss truss index maintenance. We test the proposed techniques on real datasets. The experiment results show the promise of our work.

## 1. INTRODUCTION

Truss was proposed by Jonathan Cohen in [1] in 2008 to identify cohesive subgraphs for social network analysis. The motivation originated from social structure is that if two persons are strongly tied, they should share enough common friends. As such, we have the definition of  $k$ -truss as follows:

DEFINITION 1. A  $k$ -truss in a graph  $G = (V, E)$  is a non-

trivial<sup>1</sup> connected subgraph  $G_s = (V_s, E_s)$ , where each edge in  $E_s$  is contained in at least  $k - 2$  triangles of  $G_s$ , here  $V_s \subseteq V$  and  $E_s \subseteq E$ .

Here, an edge  $(a, b)$  is contained in  $k - 2$  triangles in  $G_s$  equals that nodes  $a$  and  $b$  have  $k - 2$  common neighbors in  $G_s$ . Usually, we are interested in the maximal  $k$ -trusses, i.e. the  $k$ -trusses which are not proper subgraphs of other  $k$ -trusses, otherwise there may be too many overlapping trusses resulting from nodes combination.

Consider a social network as a graph, by identifying maximal  $k$ -trusses, we can identify a set of cohesive groups or communities. The granularity can be adjusted through the setting of  $k$ . With the identified communities, analysis can be done to extract local information from the communities and follow-up commercial strategies, such as personalized advertising [2], viral marketing [3], can then be devised based on collected information. In fact, the problem of finding cohesive subgraphs has been studied for a long time. There are many other models to identify cohesive subgraphs, e.g., clique,  $n$ -clique, quasi-clique,  $n$ -clan,  $n$ -club,  $k$ -plex and  $k$ -core. Despite many such models have been proposed, to the best of our knowledge, very little attention has been paid to maintaining the cohesive subgraphs in evolving networks. However, real networks are updated frequently. Every second, there may be people joining and leaving a network, setting up or eliminating links to others. For example, the membership of LinkedIn grows by approximately two new members every second [4]. As a result, it is imperative to maintain the discovered interesting communities (cohesive subgraphs) dynamically if possible rather than to recompute the communities from scratch each time to respond to the updating of nodes and edges of the network. In this paper, we focus on dynamically maintaining maximal trusses in evolving networks.

We first introduce other forms of cohesive subgraphs and then propose the challenge of maintaining trusses.

The basic notion is clique [5]. A clique is a subgraph where

<sup>1</sup>A trivial graph is not a 2-truss due to this non-trivial condition.

every node has links to other nodes in the subgraph. The definition of clique is too rigid. There are relaxed forms of cliques, such as quasi-clique [6, 7] and  $n$ -clique [8]. Quasi-clique relaxes the definition of clique by asking each node to connect to a certain percentage of other nodes. Quasi-clique can be defined based nodes [7] and edges [6].  $n$ -clique relaxes the distance between any two nodes in a clique from 1 to  $n$ , i.e. two nodes can be considered as connected if they are connected by at most  $n$  hops of edges.  $n$ -clan [9] is a restricted form of  $n$ -clique by further imposing a constraint on the diameter.  $n$ -club [9] removes the  $n$ -clique requirement from the  $n$ -clan, asking for only diameter constraint.  $k$ -plex [10] is also a relaxed form of clique. It requires each node in a clique of  $x$  nodes to have  $x - k$  links to other nodes. Here,  $k$  can be considered as a tolerant factor.  $k$ -plex is similar to quasi-clique. The difference is that quasi-clique asks for a percentage but  $k$ -plex asks for a number of links. The common feature of the above notions is that their computation are all NP-hard.

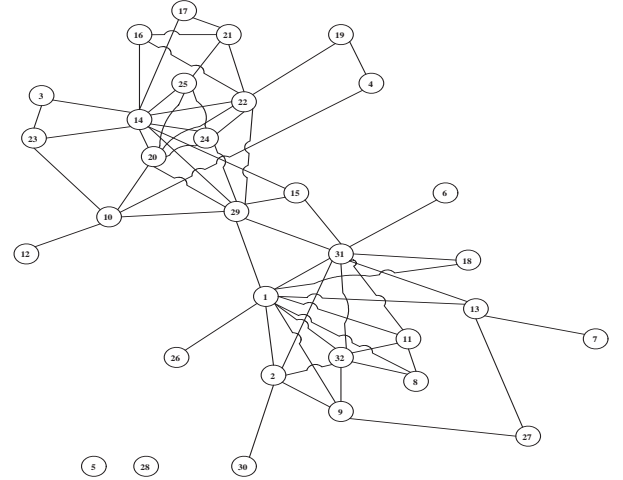
Another interesting notion, recently attracting the attention of researchers, is  $k$ -core [11].  $k$ -core is the largest subgraph where each node has its degree at least  $k$ .  $k$ -core can somehow catch a cohesive graph, because in such a graph, every node has at least  $k$  links to other nodes. The computation and maintenance of  $k$ -core is polynomial. However,  $k$ -core may not be very accurate because the existence of bridges. This has been observed in [12], where the authors use maximal induced  $k$ -connected subgraphs (or  $k$ -strong subgraphs termed in [13]) to model cohesive subgraphs. The computation of  $k$ -strong subgraphs is also expensive, though polynomial. The maintenance of  $k$ -strong subgraphs is also time-consuming [14].

Considering the computation and maintenance difficulty of  $k$ -strong subgraphs, we are expecting a model that should be less expensive to compute and maintain, but still preserve the good property of  $k$ -strong subgraphs.  $k$ -truss fits in this category. Firstly, a  $k$ -truss is both a  $(k - 1)$ -core [15] and a  $(k - 1)$ -connected subgraph, since a  $(k - 1)$ -core is a  $(k - 1)$ -connected graph. Secondly, the computation of  $k$ -truss is  $O(m^{1.5})$ , where  $m$  is the number of edges. It is low polynomial. However, it is still infeasible to recompute the trusses frequently on evolving networks. To tackle this problem, we study how to maintain  $k$ -trusses in this paper. Our result is that maintaining  $k$ -truss is  $O(|E_t|)$ , where  $E_t$  is the set of affected edges whose truss numbers need to be updated. In a large graph, we usually have  $|E_t| \ll |E|$ . This means  $k$ -truss may be a promising structure to model cohesive subgraphs in practice.

**EXAMPLE 1.** Fig. 1(a) shows the relationships of actors studied by S. Freeman and L. Freeman [1]. Fig. 1(a) shows the entire network. Fig. 1(b) shows the maximal 4-trusses. There are two in total. As the example shows, in either 4-truss, each edge is in at least 2 triangles. Fig. 1(c) shows the 3-core of the network, where each node has at least 3 neighbors. The 3-strong subgraph of the network is the same as the 3-core in this example.

Finally, we summarize the main contributions of this paper as follows:

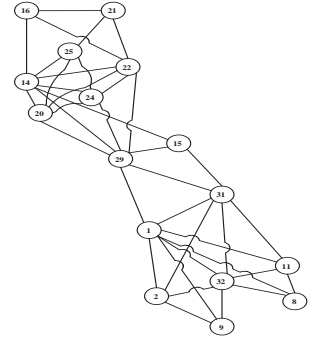
- We are the first to study the truss maintenance problem.



(a) Reciprocated Friendship and Meeting Relations



(b) 4-trusses



(c) 3-cores

**Figure 1: Examples of  $k$ -trusses and  $k$ -cores**

- We discuss a set of important truss maintenance properties by answering the following questions “which trusses may be affected”, “how much effect will an update produce”, “how far will the effect spread”.
- We devise effective truss maintenance algorithms for edge deletions and edge insertions. We prove the correctness of the algorithms and analyze the complexity of the maintenance algorithms.
- We propose to build truss indexes to speed up the evaluation of truss queries. We also introduce how to maintain the indexes.
- The truss maintenance algorithms and truss index maintenance algorithms are demonstrated as efficient by the experiments.

Here is a roadmap of this paper. In Section 2, we provide a formal definition of the problem, and introduce some necessary notations. In Section 3, we discuss the properties of truss maintenance. In Section 4, we show the algorithms of performing edge deletions or insertions. In Section 5, we show how to build and maintain truss indexes. Experiment results are shown in Section 6. Related works and conclusions are in Section 7 and Section 8 respectively.

## 2. PRELIMINARIES

**Table 1: Notations and Descriptions**

Notation	Description
$\phi(e)$	the maximal truss number of the trusses that edge $e$ resides in
$T_{\phi(e)}$	the maximal $\phi(e)$ -truss that contains edge $e$
$n(a)$	the set of neighbor nodes of node $a$
$S_{a,b}$	the common neighbors of node $a$ and $b$ : $n(a) \cap n(b)$
$E_{S_{a,b} \leftrightarrow \{a,b\}}$	the set of edges between $S_{a,b}$ and $\{a,b\}$ : $\{(n_1, n_2) : n_1 \in S_{a,b}, n_2 \in \{a,b\}\}$
$k_{max}(e)$	the maximal truss number of the edges in $E_{S_{a,b} \leftrightarrow \{a,b\}}$ : $\max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$
$k_{min}(e)$	the minimal truss number of the edges in $E_{S_{a,b} \leftrightarrow \{a,b\}}$ : $\min\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$

In this work, we consider only undirected, unweighted simple graphs. The truss definition and maximal truss definition have already been introduced in Section 1. Apparently, if an edge  $e$  is in a  $k$ -truss, it must be in a  $(k-1)$ -truss which is a supergraph of the  $k$ -truss. We define  $\phi(e)$  as the maximal truss number of the trusses that edge  $e$  can reside in. This truss is denoted as  $T_{\phi(e)}$ . We define the global support of an edge  $e$  as the number of triangles containing the edge in the graph  $G = (V, E)$ , denoted as  $sup(e, G)$ . Similarly, we define the local support of an edge  $e$  in a subgraph  $G_s = (V_s, E_s)$  ( $V_s \subseteq V$  and  $E_s \subseteq E$ ) as the number of triangles containing  $e$  in  $G_s$ , denoted as  $sup(e, G_s)$ .

LEMMA 1. *Given a graph  $G$ , we have the following:*

$$\phi(e) \leq sup(e, T_{\phi(e)}) + 2 \leq sup(e, G) + 2$$

To help with the discussions, we introduce some notions. Let  $a, b$  be two nodes,  $e = (a, b)$  be an edge to be deleted or inserted between nodes  $a$  and  $b$ ,  $n(a)$  and  $n(b)$  be the neighbor nodes of  $a$  and  $b$  respectively, we define  $S_{a,b} = n(a) \cap n(b)$  as the common neighbors of  $a$  and  $b$ , define  $E_{S_{a,b} \leftrightarrow \{a,b\}} = \{(n_1, n_2) : n_1 \in S_{a,b}, n_2 \in \{a,b\}\}$  as the set of edges between  $S_{a,b}$  and  $\{a,b\}$ . Furthermore, we define  $k_{max}(e)$  and  $k_{min}(e)$  as the maximum and minimum truss number of the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ , i.e.  $k_{max}(e) = \max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$  and  $k_{min}(e) = \min\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$ .

A notation table is given in Table 2 for easy reference. In the main part of our discussion, we consider one insertion and one deletion only, multiple insertions and deletions can be regarded as repeating single insertion and deletion.

### 3. TRUSS MAINTENANCE PROPERTIES

In this section, we discuss some properties for truss maintenance under one insertion and one deletion. Before we proceed introducing the algorithms in Section 4 to tackle the truss maintenance problem, we realize it is important and helpful to discuss some theoretical findings first. This will underpin the design of the algorithms. Given a deletion or an insertion, we believe a reader is prone to ask the following questions:

$Q_1$  Which trusses may be affected? – A set of  $k$ -trusses with different  $k$ 's may be affected. What are the affected  $k$ 's?

$Q_2$  How much effect will an update produce? – It is natural to assume that one update will affect the truss numbers of the other edges by at most 1. Is this true?

$Q_3$  How far will the effect spread? – Will the effect be bounded within an area or can the effect spread far away?

Before answering the questions raised above, we introduce the following observation. The correctness is obvious.

OBSERVATION 1. *After an insertion, the truss number of any edge will not decrease; after a deletion, the truss number of any edge will not increase.*

#### 3.1 How much effect will an update produce?

We start with the “how much” question ( $Q_2$ ) first, because the answer is easier to understand and will be used to answer the “which” question ( $Q_1$ ). The answer to  $Q_2$  is given in Lemma 2.

LEMMA 2. *After one deletion or insertion, if an edge is affected, the truss number is affected by at most 1, i.e. increased by 1 or decreased by 1.*

PROOF. We prove the deletion first and then the insertion.

Let  $e$  be an edge to be deleted and its truss number be  $\phi(e)$ , firstly any edge  $e'$  with truss number  $\phi(e')$  more than  $\phi(e)$  is not affected, because  $e$  is not in the  $\phi(e')$ -truss since  $\phi(e') > \phi(e)$ , therefore deleting  $e$  will not affect any edge in the  $\phi(e')$ -truss, i.e.  $e'$  is still in the  $\phi(e')$ -truss. Secondly, this time, let  $e'$  be any edge in the  $\phi(e')$ -truss  $T_{\phi(e')}$  with  $\phi(e') \leq \phi(e)$ , after deleting  $e$ , the global and local support of  $e'$  will reduce by at most 1 respectively (both may remain unchanged), because  $e$  and  $e'$  only participate together in at most one triangle. As a result, after deleting  $e$ ,  $e'$  is guaranteed to be in the  $(\phi(e') - 1)$ -truss. In both cases, the effect is not more than 1.

The insertion follows the deletion, and can be proved by contradiction. Suppose inserting  $e$  increases the truss number of an edge  $e'$  by more than 1, then for the updated graph, deleting  $e$  will cause the truss number of  $e'$  to decrease by more than 1. This contradicts the first (deletion) part of the proof.  $\square$

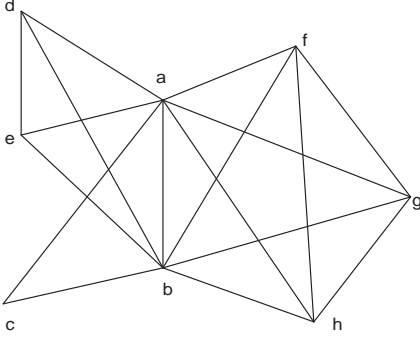
#### 3.2 Which trusses may be affected?

The answer to this “which” question is given in Theorem 1 (for deletion) and Theorem 2 (for insertion).

THEOREM 1. *For deletion, let  $e = (a, b)$  be the to-be-deleted edge, (a) if  $S_{a,b} = \emptyset$  or  $k_{min}(e) > \phi(e)$ , no other edges are affected; (b) if  $S_{a,b} \neq \emptyset$  and  $k_{min}(e) \leq \phi(e)$ , the possibly affected trusses must have truss numbers within the range  $[k_{min}(e), \phi(e)]$ .*

PROOF. For case (a), if  $S_{a,b} = \emptyset$ , it is obvious that no other edges are affected, because no triangles contain  $(a, b)$ ; if  $k_{min}(e) > \phi(e)$ , no other edges are affected, because any edge  $e'$  with truss number  $\phi(e')$  more than  $\phi(e)$  is not affected. The reason is that  $e = (a, b)$  is not in the  $\phi(e')$ -truss containing  $e'$  since  $\phi(e') > \phi(e)$ , therefore deleting  $e$  will not affect  $e'$ .

For case (b),  $S_{a,b} \neq \emptyset$  and  $k_{min}(e) \leq \phi(e)$ , (1) firstly, again, as proved in case (a), any edge  $e'$  with truss number



**Figure 2: A  $K_3$ , a  $K_4$  and a  $K_5$  are joined at  $(a,b)$**

$\phi(e')$  more than  $\phi(e)$  is not affected. (2) next, we prove that any edge with truss number less than  $k_{\min}(e)$  will not be affected. Let  $e'$  be an edge with truss number  $\phi(e') < k_{\min}(e)$ , then according to Lemma 2, after deleting  $e$ , all edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  remain in the  $\phi(e')$ -truss, because the truss number of an edge can be affected by at most 1 and the minimum truss number directly affected by deleting  $e$  is  $k_{\min}(e)$ . As a result, the updated  $\phi(e')$ -truss is the same as the old  $\phi(e')$ -truss but without the deleted edge  $e$ , and thus the edge  $e'$  remains in the  $\phi(e')$ -truss. As a result, combining case (1) and case (2), the possibly affected trusses are in the range of  $[k_{\min}(e), \phi(e)]$ .  $\square$

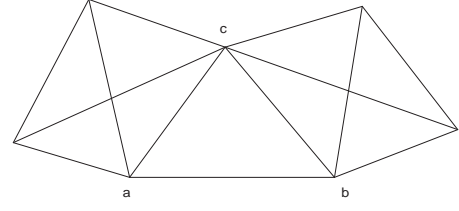
Moreover, we show that the range  $[k_{\min}(e), \phi(e)]$  is tight with an example. Let  $K_n$  denote a complete graph with  $n$  nodes, suppose a graph is joined by a set of complete graphs  $K_k, K_{k-1}, \dots, K_t$  ( $t \geq 3$ ) on an edge  $e$ , i.e.  $e$  is the only shared edge between  $K_k, K_{k-1}, \dots, K_t$ , in such a graph, all edges in the subgraph  $K_k$  including  $e$  have the truss number  $k = \phi(e)$ , all edges in the subgraph  $K_{k-1}$  except  $e$  have the truss number  $k-1$ , ..., all edges in the subgraph  $K_t$  except  $e$  have the truss number  $t$ . Apparently, in this example,  $k_{\min}(e) = t$ . If we delete the edge  $e$ , we can find that the truss numbers of all other edges will decrease by 1. These edges originally have the truss numbers ranging from  $t = k_{\min}(e)$  to  $k = \phi(e)$ , therefore the range  $[k_{\min}(e), \phi(e)]$  is tight. Fig. 2 is one such example. A  $K_3$ , a  $K_4$  and a  $K_5$  are joined at edge  $(a,b)$ . If we delete  $(a,b)$ , all the other edges are affected and they originally have the truss range  $[3, 5]$ . Here, for the edge  $(a,b)$ ,  $k_{\min}((a,b)) = 3$  and  $\phi((a,b)) = 5$ . The affected range  $[3, 5]$  is tight in this example.

Before presenting Theorem 2 for the insertion, we introduce Lemma 3 and Lemma 4 first to support the proof of Theorem 2.

**LEMMA 3.** *After inserting an edge  $e$ , if the truss number of an edge  $e'$  increases from  $\phi(e')$  to  $\phi(e') + 1$ , then the inserted edge  $e$  must be in the  $(\phi(e') + 1)$ -truss containing  $e'$  in the updated graph.*

**PROOF.** By contradiction, if the inserted edge  $e$  is not in the  $(\phi(e') + 1)$ -truss containing  $e'$  in the updated graph, then deleting  $e$  will not reduce the truss number of  $e'$ . This contradicts with the assumption that after inserting  $e$ , the truss number of  $e'$  increases from  $\phi(e')$  to  $\phi(e') + 1$ .  $\square$

**LEMMA 4.** *For any edge  $e = (a,b)$  in a graph, we have*



**Figure 3: An example showing  $\phi((a,b)) \geq \min\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$  does not hold.**

the following:

$$\begin{cases} \phi(e) = 2 & (E_{S_{a,b} \leftrightarrow \{a,b\}} = \emptyset) \\ \phi(e) \leq \max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\} & (E_{S_{a,b} \leftrightarrow \{a,b\}} \neq \emptyset) \end{cases}$$

**PROOF.** If  $E_{S_{a,b} \leftrightarrow \{a,b\}} = \emptyset$  (i.e.  $S_{a,b} = n(a) \cap n(b) = \emptyset$ ), the truss number of  $(a,b)$  is 2. The lemma holds naturally. If  $E_{S_{a,b} \leftrightarrow \{a,b\}} \neq \emptyset$ , assume  $\phi(e) > \max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$ , then we have that edge  $(a,b)$  is in a  $\phi(e)$ -truss and all the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  are not in the  $\phi(e)$ -truss. Therefore, edge  $(a,b)$  does not have supporting triangles in the  $\phi(e)$ -truss, so  $\phi(e)$  must be 2. This contradicts with the assumption  $\phi(e) > \max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$  in which it is apparent that  $\max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\} \geq 3$  if  $E_{S_{a,b} \leftrightarrow \{a,b\}} \neq \emptyset$ , so  $\phi(e) \leq \max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$  when  $E_{S_{a,b} \leftrightarrow \{a,b\}} \neq \emptyset$ .  $\square$

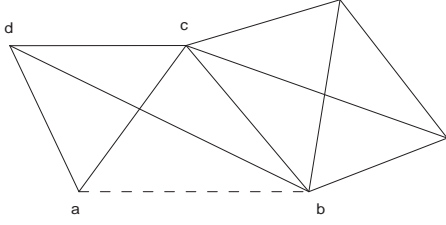
Lemma 4 says the maximum truss number of edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  is an upper bound of the truss number of  $(a,b)$ . Similarly, readers may wonder “do we have the minimum truss number of the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  as a lower bound of the truss number of  $(a,b)$  when  $E_{S_{a,b} \leftrightarrow \{a,b\}} \neq \emptyset$ ”, i.e. “do we have  $\phi(e) \geq \min\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$  when  $E_{S_{a,b} \leftrightarrow \{a,b\}} \neq \emptyset$ ”? The answer is no. We illustrate this through an example in Fig. 3, where  $\phi((a,b)) = 3$ ,  $S_{a,b} = n(a) \cap n(b) = \{c\}$ ,  $E_{S_{a,b} \leftrightarrow \{a,b\}} = \{(a,c), (b,c)\}$ ,  $\min\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\} = \min\{\phi((a,c)) = 4, \phi((b,c)) = 4\} = 4$ . In this example,  $\phi((a,b)) < \min\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\}$ .

Now we are ready to state the following theorem.

**THEOREM 2.** *For insertion, let  $e = (a,b)$  be the newly inserted edge, (a) if  $S_{a,b} = \emptyset$  or  $k_{\min}(e) > |S_{a,b}| + 1$ , no other edges are affected; (b) if  $S_{a,b} \neq \emptyset$  and  $k_{\min}(e) \leq |S_{a,b}| + 1$ , the possibly affected trusses must have truss numbers within the range  $[k_{\min}(e), \min(|S_{a,b}| + 1, k_{\max}(e))]$ .*

**PROOF.** For case (a), if  $S_{a,b} = \emptyset$ , obviously no other edges are affected; if  $k_{\min}(e) > |S_{a,b}| + 1$ , no other edges are affected because any edge  $e'$  with  $\phi(e') > |S_{a,b}| + 1$  is not affected. Assume the truss of  $e'$  is affected and increased to  $\phi(e') + 1$ , according to Lemma 3,  $(a,b)$  will be in the  $(\phi(e') + 1)$ -truss with the truss number larger than  $|S_{a,b}| + 2$ . However, it is obvious that  $|S_{a,b}|$  is the global support of  $(a,b)$ , so  $|S_{a,b}| + 2$  is an upper bound of  $(a,b)$ 's truss number. A contradiction appears.

For case (b),  $S_{a,b} \neq \emptyset$  and  $k_{\min}(e) \leq |S_{a,b}| + 1$ , if an edge  $e'$  is affected, then we must have (1)  $\phi(e') \leq |S_{a,b}| + 1$ , (2)  $\phi(e') \leq k_{\max}(e)$  and (3)  $\phi(e') \geq k_{\min}(e)$ . We now prove each part respectively. (1) Firstly,  $\phi(e') \leq |S_{a,b}| + 1$  is proved in case (a). (2) Secondly, any edge  $e'$  with  $\phi(e') > k_{\max}(e)$  is



**Figure 4:**  $k_{max}((a,b)) = 4$ ,  $k_{min}((a,b)) = 3$ ,  $S_{a,b} = \{c,d\}$ ,  $k_{min}((a,b)) \leq |S_{a,b}| + 1 < k_{max}((a,b))$ , so the affected truss range is  $[k_{min}((a,b)), |S_{a,b}| + 1]$ .

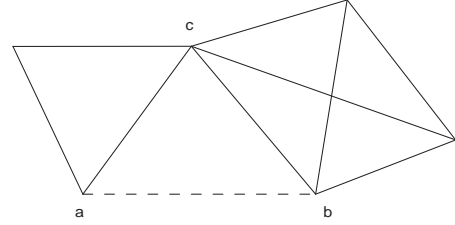
not affected. We prove by contradiction. Assume the truss number of  $e'$  is increased to  $\phi(e') + 1$ , according to Lemma 3, the truss number of the inserted edge  $e = (a,b)$ ,  $\phi(e)$ , is at least  $\phi(e') + 1$  in the updated graph, so we have  $\phi(e) > k_{max}(e) + 1$ . On the other hand, according to Lemma 2, in the updated graph,  $\max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\} \leq k_{max}(e) + 1$ . Furthermore, according to Lemma 4, in the updated graph,  $\phi(e) \leq \max\{\phi(e') : e' \in E_{S_{a,b} \leftrightarrow \{a,b\}}\} \leq k_{max}(e) + 1$ . There is a contradiction. (3) Finally, any edge  $e'$  with  $\phi(e') < k_{min}(e)$  is not affected. Since  $\phi(e') < k_{min}(e)$ , all the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  are in a  $(\phi(e') + 1)$ -truss. As a result, there is no new edges adding into the  $(\phi(e') + 1)$ -truss except the newly inserted one, because all the directly affected edges  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  are all already in the  $(\phi(e') + 1)$ -truss. The unchangingness of any  $(\phi(e') + 1)$ -truss with  $\phi(e') < k_{min}(e)$ , together with Observation 1, implies that current  $\phi(e')$ -truss edges remain as  $\phi(e')$ -truss edges.  $\square$

Then, let us inspect the tightness of  $[k_{min}(e), \min\{|S_{a,b}| + 1, k_{max}(e)\}]$ . Similar to the deletion, suppose a graph is joined by a set of complete graphs  $K_k, K_{k-1}, \dots, K_t (t \geq 3)$  on an edge  $e$ , but now  $e$  was deleted, and we want to insert  $e$  back into the  $e$ -deleted graph. Then the affected trusses will have truss ranges from  $t - 1$  to  $k - 1$ .  $t - 1$  and  $k - 1$  are the  $k_{min}(e)$  and  $k_{max}(e)$  of the to-be-inserted edge  $e$  in the  $e$ -deleted graph. Sometimes,  $k_{max}(e)$  may be larger than  $|S_{a,b}| + 1$ . In such cases,  $[k_{min}(e), |S_{a,b}| + 1]$  is the affected truss range. See Fig. 4 for an example,  $(a,b)$  is the edge to be inserted,  $k_{max}((a,b))$  is 4 on the edge  $(b,c)$ ,  $k_{min}((a,b))$  is 3 on the edges  $(a,d)$ ,  $(b,d)$  and  $(a,c)$ ,  $S_{a,b} = \{c,d\}$ , so  $k_{min}((a,b)) = |S_{a,b}| + 1 = 3 < k_{max}((a,b))$ . The affected truss range is  $[k_{min}((a,b)), |S_{a,b}| + 1]$ , which is  $[3, 3]$ . In this example, the affected edges are  $(a,d)$ ,  $(b,d)$ ,  $(a,c)$  and  $(c,d)$ . All of them originally have truss number 3. In the next example,  $k_{min}((a,b)) > |S_{a,b}| + 1$ , see Fig. 5. Again,  $(a,b)$  is the to-be-inserted edge,  $S_{a,b} = \{c\}$ ,  $k_{max}((a,b)) = 4$  on the edge  $(b,c)$ ,  $k_{min}((a,b)) = 3$  on the edge  $(a,c)$ ,  $|S_{a,b}| + 1 = 2 < k_{min}((a,b))$ . No existing edges are affected when we insert  $(a,b)$ .

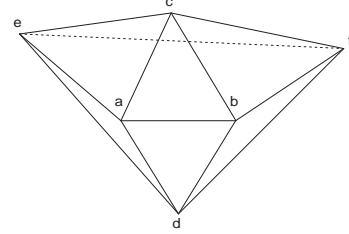
Theorem 1 and 2 tell us that, for an update, we do not need to inspect all the edges in the graph, instead we only need to inspect a subset of edges whose truss numbers are within a certain range.

### 3.3 How far will the effect spread?

From the above discussions, we find that the truss number of an edge is affected because the local support of the edge is changed. When we delete or insert an edge  $(a,b)$ , recall that  $S = n(a) \cap n(b)$ , the set of edges that are directly



**Figure 5:**  $k_{max}((a,b)) = 4$ ,  $k_{min}((a,b)) = 3$ ,  $S_{a,b} = \{c\}$ ,  $k_{min}((a,b)) > |S_{a,b}| + 1$ , so no other edges are affected.



**Figure 6:** The effect may spread

affected is  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ , because  $(a,b)$  forms triangles with the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ . As a result, the global support of the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  will change, and this may lead to changes of their local support, eventually the truss numbers of these edges may change. Furthermore, the effect may spread to the neighbor edges of the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ . In this section, we will discuss how far the effect will spread. For simplicity, we only discuss deletion in all the examples in this section. The effect of insertion is similar. (one can simply insert the deleted edge back and see the impact of an insertion in the following examples.)

Firstly, we show that the effect may spread indeed. It is natural to recognize that the truss numbers of edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  may be affected, but could other edges be affected, such as the edges incident on node  $a$  or  $b$  but not in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ ? The answer is yes. We give an example in Fig. 6. The graph is a 4-truss. Every edge is in two triangles. If we delete  $(a,b)$ , directly we will have  $(a,c)$ ,  $(a,d)$ ,  $(b,c)$ ,  $(b,d)$  having global support 1, and thus these four edges cannot be in a 4-truss. Further, we find that there is no 4-truss in the updated graph. The truss numbers of all the other edges are reduced from 4 to 3, such as  $(a,e)$ ,  $(b,f)$ , though they do not belong to  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ .

Next, we give another example to show that the effect can be spread a long way away (summarized as Lemma 5). In Fig. 7, there is an unlimited graph piling by the unit shown in the bottom right corner. In such a graph, every edge is in two triangles and every edge has truss number 4. If we delete edge  $(a,b)$ , the first set of edges whose truss numbers are affected is  $(a,p_2)$ ,  $(b,p_2)$ ,  $(a,q_3)$ ,  $(b,q_3)$ . Each of these four edges will have only one supporting triangle, so they will no longer be in the 4-truss. Then, the effect will spread towards four directions leading to the truss number of every edge decreased by 1.

LEMMA 5. *There is no guarantee that the update effect will end after spreading a fixed number of steps.*



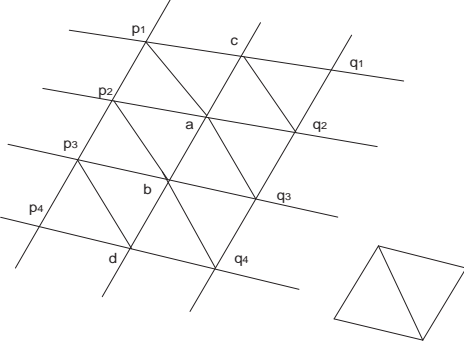


Figure 7: The effect may spread very far

## 4. ALGORITHMS

Based on the properties discussed in Section 3, we design maintenance algorithms for deleting an edge and inserting an edge.

### 4.1 Deleting an edge

Inspired by the answers to the questions “*which trusses may be affected*” and “*how far will the effect spread*” in Section 3, we can devise an outward inspection based truss maintenance approach for edge deletion. Firstly, we start from the to-be-deleted edge  $(a, b)$  and check which neighbor edges will be directly affected if we delete  $(a, b)$ . These directly affected edges must be in the edge set  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ , because other edges (the edges not in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ ) do not participate in triangles together with  $(a, b)$ , and thus they are definitely not affected at the moment, although we may find that they are affected at a later time. After inspecting the set  $E_{S_{a,b} \leftrightarrow \{a,b\}}$ , we may find some edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  are affected while the others are not. Then, we continue inspecting the neighbor edges of the affected edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  and repeat the process until no newly affected edges are found. During the process, only the edges with truss numbers in the range  $[k_{\min}((a, b)), \phi((a, b))]$  need to be examined. This is guaranteed by Theorem 1. Algorithm 1 gives the maintenance steps.

We now explain Algorithm 1 in detail. Line 1 deletes the edge  $(a, b)$ . Line 2 puts the pruned edge set  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  into a queue, since this is the set of edges whose truss numbers may be directly affected by deleting  $(a, b)$ . While the queue is not empty (line 3), i.e. there are still some edges that may be affected directly or indirectly waiting to be inspected, we take an edge from the queue (line 4), and check whether the truss number of the edge should be decreased (line 5). The first condition “ $(v_1, v_2)$  is not marked” means if the truss number of an edge has been decreased before (see line 7) we do not need to decrease the truss number again, because according to Lemma 2 we only need to decrease the truss number once. The second condition of line 5 checks whether the edge  $(v_1, v_2)$  has enough local support with respect to the truss number  $\phi((v_1, v_2))$ . If not, i.e. “ $localSupport((v_1, v_2), \phi((v_1, v_2))) < \phi((v_1, v_2)) - 2$ ” is true, we decrease the truss number of  $(v_1, v_2)$  (line 6), mark the edge  $(v_1, v_2)$  (line 7) and put the pruned edges in  $E_{S_{v_1, v_2} \leftrightarrow \{v_1, v_2\}}$  with truss numbers in  $[k_{\min}((a, b)), \phi((a, b))]$  into the queue (line 8). The truss numbers of the edges in the set  $E_{S_{v_1, v_2} \leftrightarrow \{v_1, v_2\}}$  may be affected because the truss num-

---

**Algorithm 1** An outwards inspection based truss maintenance algorithm for edge deletion

---

**Input:** a graph  $G = (V, E)$  with truss numbers associated with its edges, a to-be-deleted edge  $(a, b)$ ;

**Output:**  $G = (V, E - \{(a, b)\})$  with updated truss numbers associated with its edges;

```

1: delete the edge  $(a, b)$ ;
2: put the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  with truss numbers in
    $[k_{\min}((a, b)), \phi((a, b))]$  (Theorem 1) into a queue;
3: while the queue is not empty do
4:   take an edge  $(v_1, v_2)$  from the queue;
5:   if  $(v_1, v_2)$  is not marked  $\wedge$ 
       $localSupport((v_1, v_2), \phi((v_1, v_2))) < \phi((v_1, v_2)) - 2$ 
      then
6:     decrease the truss number of  $(v_1, v_2)$ ;
7:     mark the edge  $(v_1, v_2)$ ;
8:     put the edges in  $E_{S_{v_1, v_2} \leftrightarrow \{v_1, v_2\}}$  with truss num-
       bers in  $[k_{\min}((a, b)), \phi((a, b))]$  (Theorem 1) into the
       queue;
9:   end if
10: end while
11: start from the edge  $(a, b)$ , do a depth-first or breadth-
    first search to unmark all the marked edges;
    {This is to guarantee the correctness of processing the
     next deletion or insertion.}
12: return  $G$ ;
```

**Step 5:**  $localSupport((v_1, v_2), k)$

**Input:** an edge  $(v_1, v_2)$  and a truss number  $k$ ;

**Output:** the local support of the edge  $(v_1, v_2)$  with respect to the truss number  $k$ ;

```

1:  $lSupport \leftarrow 0$ ;
2: for each node  $v \in n(v_1) \cap n(v_2)$  do
3:   if  $\phi((v, v_1)) \geq k \wedge \phi((v, v_2)) \geq k$  then
4:      $lSupport \leftarrow lSupport + 1$ ;
5:   end if
6: end for
7: return  $lSupport$ ;
```

---

ber of the edge  $(v_1, v_2)$  is affected. On the contrary, if the condition “ $localSupport((v_1, v_2), \phi((v_1, v_2))) < \phi((v_1, v_2)) - 2$ ” is false, no action is required. We simply go on to inspect the next edge in the queue, i.e. start a new loop in the while clause from line 4. Finally, after we empty the queue, the truss numbers of the edges in  $G$  have been updated. Line 11 is a post-processing step that prepares for the next update operation.

We now explain the function  $localSupport((v_1, v_2), k)$ . In order to compute the local support of the edge  $(v_1, v_2)$  with respect to the truss number  $k$ , we need to count how many triangles containing the edge  $(v_1, v_2)$  also have the other two edges with truss numbers no less than  $k$ . This equals to count how many such node  $v \in S_{v_1, v_2} = n(v_1) \cap n(v_2)$  having both  $\phi((v, v_1))$  and  $\phi((v, v_2))$  no less than  $k$ .

In the following, we will prove Algorithm 1 is correct and analyze the algorithm complexity.

**LEMMA 6.** *Algorithm 1 is correct. The complexity is  $O(|E_l|)$ , where  $|E_l|$  is the number of edges whose local support are affected and  $|E_l| < |E|$ .*

**PROOF.** The first possibly affected set of edges are put into the queue in line 2. The following possibly affected

edges are put into the queue in line 8. All possibly affected edges are put into the queue by Algorithm 1. When an edge  $(v_1, v_2)$  is taken from the queue to be inspected, the source affected edge which leads  $(v_1, v_2)$  to be put into the queue has been updated (the truss number is decreased). This guarantees the current computed local support of  $(v_1, v_2)$  is an upper bound of the local support in the final updated graph. Therefore, it is safe to decrease the truss number of  $(v_1, v_2)$  if the current local support is already less than  $\phi((v_1, v_2)) - 2$ . It is also guaranteed that the truss number can be decreased only once.

Complexity of Algorithm 1: Assume the graph  $G = (V, E)$  is stored in an adjacency list, we analyze the neighbor set intersection  $n(a) \cap n(b)$  first. By using a hash table,  $n(a) \cap n(b)$  can be done in  $O(|n(a)| + |n(b)|)$  time. This is accomplished by firstly putting the edges in  $n(a)$  into the hash table, and then hash-check the edges in  $n(b)$ . (An inferior method using nested-loop checks requires  $O(|n(a)||n(b)|)$  time.) Back to our discussion on Algorithm 1, line 1, line 2 and line 11 have complexity  $O(1)$ ,  $O(|n(a)| + |n(b)|)$  and  $O(|E|)$  respectively. These are dominated by the while clause, line 3 to line 10. In the while clause, the critical step is line 5 and line 8. For line 5, the function  $localSupport()$  has complexity  $O(|n(v_1)| + |n(v_2)|)$ , because line 2 in the function  $localSupport()$  has the complexity  $O(|n(v_1)| + |n(v_2)|)$  and all the other steps in the function  $localSupport()$  have the complexity  $O(1)$ . For line 8, the complexity is  $O(|n(v_1)| + |n(v_2)|)$  as well. As a result, the complexity of Algorithm 1 is  $O(n_q \cdot deg_{max})$ , where  $n_q$  is the number of edges ever put into the queue and  $deg_{max}$  is the maximum degree of the nodes in the graph. To analyze the bounds: in a massive graph,  $deg_{max}$  can be considered as a constant. We also have  $n_q = O(|E_l|)$ , where  $|E_l|$  is the number of edges whose local support is affected. The reason is that an edge  $(a, b)$  will be put into the queue at most  $2|S_{a,b}|$  times and  $2|S_{a,b}| < 2|deg_{max}|$ , so  $n_q < 2|deg_{max}||E_l|$ . As a result, we have  $n_q = O(|E_l|)$ . As the affected edges are far less than the number of edges in the graph ( $|E_l| \ll |E|$ ), the maintenance algorithm is very efficient. The cost of the maintenance algorithm is linear to the number of affected edges, which is  $O(E_l)$ .

□

## 4.2 Inserting an edge

When an edge is inserted, the truss maintenance includes two parts. Firstly, according to Observation 1 and Lemma 2, for some existing edges, the truss numbers should be increased by 1, and all the other existing edges keep their truss numbers unchanged. Secondly, for the newly inserted edge, the truss number needs to be determined.

### 4.2.1 Different from deleting an edge

One may think to design an approach for inserting an edge similar to the approach designed for deleting an edge. The idea of the hypothesized algorithm is as follows: we start from the to-be-inserted edge. Firstly, we find which neighbor edges will have their global support increased by 1 and put them into a queue. Let  $(v_1, v_2)$  be one of the edges in the queue and let its truss number be  $\phi((v_1, v_2))$ , we then check whether the local support of  $(v_1, v_2)$  with respect to the truss number  $\phi((v_1, v_2)) + 1$  is no less than  $\phi((v_1, v_2)) - 1$ . If so, it means that  $(v_1, v_2)$  has local support at least  $\phi((v_1, v_2)) - 1$ . We then increase the truss number

of  $(v_1, v_2)$  by 1 and go on to inspect the neighbor edges in  $E_{S_{v_1, v_2} \leftrightarrow \{v_1, v_2\}}$ .

The hypothesized algorithm has a similar paradigm as the deletion counterpart. However, it is not correct. The problem is that, we may miss to update some edges. In other words, after an insertion, some edges may have truss numbers increased from  $k$  to  $(k + 1)$ , but we cannot find them using the algorithm. It is safe to increase the truss number of an edge  $(v_1, v_2)$  if the local support of the edge in the current  $(\phi((v_1, v_2)) + 1)$ -truss is already no less than  $\phi((v_1, v_2)) - 1$ , however, when we check the condition " $localSupport((v_1, v_2), \phi(v_1, v_2) + 1) \geq \phi((v_1, v_2)) - 1$ ", some edges which should finally be put into the  $(\phi((v_1, v_2)) + 1)$ -truss have not been found, so at this time, the local support may be underestimated, as a result, we may miss to update some  $(\phi((v_1, v_2)) + 1)$ -truss edges. In the following section, we introduce the correct approaches to update truss numbers in case of edge insertions.

### 4.2.2 Algorithm

For truss maintenance under insertion, an intuitive, naive but correct approach is: starting from the inserted edge  $e = (a, b)$ , do a depth-first or breadth-first search to find the  $k_{min}(e)$ -truss. During the process, eliminate the edges with truss numbers more than  $\min(|S_{a,b}| + 1, k_{max}(e))$ . As a result, we can obtain a pruned  $k_{min}(e)$ -truss. In the end, we add the edge  $e$  into the pruned  $k_{min}(e)$ -truss and do a truss decomposition. The truss numbers of all the edges will then be up-to-date. The correctness of this naive method is obvious, guaranteed by Theorem 2. However, the drawback of the naive approach is that it ignores the effect actually started from the inserted edge. In an extreme case, if we discover that the first possibly affected set of edges  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  are actually not affected after inserting edge  $e$ , we can stop the maintenance process right away, because no other edges will be affected. We only need to determine the truss number for the newly inserted edge  $e$ . Along this line, we design a mark-and-verify approach for truss maintenance under an edge insertion.

The idea of mark-and-verify approach is that we determine whether the truss number of an edge should be increased in two steps. In the mark step, we try to find which edges may have their truss numbers be increased. An edge is marked if the truss number of the edge is possible to be increased by 1. As the inspection goes on, once we find an edge cannot be updated, we go on checking the neighbor marked edges to see whether there are other marked edges are affected, i.e. do not need to be updated as well. The marking steps and the verifying steps are intertwined.

The mark-and-verify algorithm is given in Algorithm 2. We now explain in detail. Firstly, we insert the edge  $(a, b)$  and put the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  confined by Theorem 2 into the queue (line 1 and 2). While the queue is not empty (line 3), we take an edge from the queue (line 4), and check whether the condition " $localSupport2((v_1, v_2), \phi(v_1, v_2)) \geq \phi((v_1, v_2)) - 1$ ". If so, it means the edge  $(v_1, v_2)$  may have its truss number increased by 1, because  $(v_1, v_2)$  has enough local support at the moment. (Function  $localSupport2()$  will be introduced in detail in the next paragraph.) As a result, if  $(v_1, v_2)$  has not been identified as possibly-affected, i.e.  $(v_1, v_2)$  is not marked (line 6), we mark the edge  $(v_1, v_2)$  and put the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  confined by Theorem 2 into the queue (line 7 and 8). If  $(v_1, v_2)$  is marked, nothing needs to be done. We just leave  $(v_1, v_2)$  as marked (possibly-

---

**Algorithm 2** A mark-and-verify truss maintenance algorithm for edge insertion

---

**Input:** a graph  $G = (V, E)$  with truss numbers associated with its edges, a to-be-inserted edge  $(a, b)$

**Output:**  $G = (V, E \cup \{(a, b)\})$  with updated truss numbers associated with its edges

```

1: insert edge  $(a, b)$ ;
2: put the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  confined by Theorem 2 into
   a queue;
3: while the queue is not empty do
4:   take an edge  $(v_1, v_2)$  from the queue;
5:   if  $localSupport2((v_1, v_2), \phi(v_1, v_2)) \geq \phi(v_1, v_2) - 1$ 
     then
6:     if  $(v_1, v_2)$  is not marked then
7:       mark the edge  $(v_1, v_2)$ ;
8:       put the edges in  $E_{S_{v_1, v_2} \leftrightarrow \{v_1, v_2\}}$  confined by The-
         orem 2 into the queue;
9:     end if
10:  else
11:    if  $(v_1, v_2)$  is marked then
12:      unmark the edge  $(v_1, v_2)$ ;
13:      record the truss number of  $(v_1, v_2)$  as unchanged,
        i.e.  $unchanged((v_1, v_2)) = true$ ;
14:      put the edges in  $E_{S_{v_1, v_2} \leftrightarrow \{v_1, v_2\}}$  confined by The-
        orem 2 into the queue;
15:    end if
16:  end if
17: end while
18: start from the edge  $(a, b)$ , do a depth-first or breadth-
    first search: for all marked edges, unmark them and
    increase the truss numbers by 1;
19: compute the truss number of the inserted edge  $(a, b)$ ;
20: return  $G$ ;

```

---

affected). On the contrary, if “ $localSupport2((v_1, v_2), \phi(v_1, v_2)) < \phi(v_1, v_2) - 1$ ”, it means the edge  $(v_1, v_2)$  will keep its truss number unchanged, because it does not have enough local support with respect to the truss number  $\phi(v_1, v_2) + 1$ . As a result, if  $(v_1, v_2)$  is marked (identified as possibly-affected previously) in line 11, we unmark the edge  $(v_1, v_2)$ , record the truss number of  $(v_1, v_2)$  as unchanged and put the edges in  $E_{S_{a,b} \leftrightarrow \{a,b\}}$  confined by Theorem 2 into the queue (line 12, 13 and 14). Finally, in line 18, for all the left marked edges, we increase their truss numbers by 1, because these are the affected edges. We also need to unmark them eventually to prepare for the maintenance of the next update operation. In line 19, we compute the truss number of the newly inserted edge  $(a, b)$ .

We now explain the function  $localSupport2()$ . The function takes an edge  $(v_1, v_2)$  and a truss number  $k$  as input. Here,  $k = \phi(v_1, v_2)$ . The function returns an upper-bound of the local support  $sup((v_1, v_2), T_{\phi(v_1, v_2)+1})$ , where  $sup((v_1, v_2), T_{\phi(v_1, v_2)+1})$  is the support of the edge  $(v_1, v_2)$  in a  $(\phi(v_1, v_2) + 1)$ -truss. In the beginning, all the edges with truss number  $k$  have the possibility to become  $(k + 1)$ -truss edges. As a result,  $localSupport2((v_1, v_2), k)$  returns the highest bound of  $sup((v_1, v_2), T_{\phi(v_1, v_2)+1})$ . As the inspection goes on, some edges are found to be not affected, e.g. there may be a node  $v \in n(v_1) \cap n(v_2)$  having  $unchanged((v, v_1))$  changed to *true* by Step 13 in Algorithm 2. In such a case, the function  $localSupport2((v_1, v_2), k)$  will return a smaller

---

**Algorithm 3**  $localSupport2((v_1, v_2), k)$

---

**Input:** an edge  $(v_1, v_2)$  and a truss number  $k$ ;

**Output:** the upperbound local support of  $(v_1, v_2)$  with respect to the truss number  $k$ ;

```

1:  $lSupport \leftarrow 0$ ;
2: for each node  $v \in n(v_1) \cap n(v_2)$  do
3:   if  $\phi((v, v_1)) \geq k \wedge unchanged((v, v_1)) == false \wedge$ 
      $\phi((v, v_2)) \geq k \wedge unchanged((v, v_2)) == false$  then
4:      $lSupport \leftarrow lSupport + 1$ ;
5:   end if
6: end for
7: return  $lSupport$ ;

```

---

value on the next call. As more edges are found to be not affected, the value returned by the function will become smaller. Once the function  $localSupport2((v_1, v_2), k)$  returns a value that is less than  $k - 1$ , we can conclude that  $(v_1, v_2)$  will not have its truss number increased by 1. Lemma 7 guarantees the conclusion.

**LEMMA 7.** *Let edge  $(a, b)$  have the truss number at least  $k$ , if the local support of  $(a, b)$  in the  $k$ -truss containing  $(a, b)$  is smaller than  $(k - 1)$ , then edge  $(a, b)$  cannot be in a  $(k + 1)$ -truss.*

**PROOF.** By contradiction, assume  $(a, b)$  can be in a  $(k + 1)$ -truss, then a necessary condition is that the edge should have local support at least  $k - 1$  in the  $(k + 1)$ -truss. Since a  $(k + 1)$ -truss is also a  $k$ -truss, it implies that  $(a, b)$  must have at least  $k - 1$  support in the  $k$ -truss containing  $(a, b)$ . A contradiction appears.  $\square$

In the following, we prove Algorithm 2 is correct and analyze the algorithm complexity.

**LEMMA 8.** *Algorithm 2 is correct. The complexity is  $O(|E_l|)$ , where  $|E_l|$  is the number of edges whose local support are affected and  $|E_l| < |E|$ .*

**PROOF.** The first possibly-affected set of edges are put into the queue in line 2. For the other edges, the local support condition “ $localSupport2((v_1, v_2), \phi(v_1, v_2)) \geq \phi(v_1, v_2) - 1$ ” and line 6-8 guarantee that all possibly-affected edges will be put into the queue. “ $localSupport2((v_1, v_2), \phi(v_1, v_2)) < \phi(v_1, v_2) - 1$ ” and line 11-14 guarantee that once an edge is found to be not affected. The effect will be spread to the full set of other edges.

Complexity of Algorithm 2: similar to Algorithm 1, the dominating step of Algorithm 2 is the while clause, from line 3 to line 17. In the while clause, the critical steps are line 5, line 8 and line 14. All of them have complexity  $O(|n(v_1)| + |n(v_2)|)$ . Again, the complexity of Algorithm 2 is  $O(n_q \cdot deg_{max})$ . Different from the deletion case,  $n_q$  is bounded by  $4|deg_{max}||E_l|$  because the edges in  $E_{S_{v_1, v_2} \leftrightarrow \{v_1, v_2\}}$  will be put into the queue when we both mark and unmark the edge  $(v_1, v_2)$ , while for the deletion algorithm, we only put the edges in  $E_{S_{v_1, v_2} \leftrightarrow \{v_1, v_2\}}$  into the queue when we mark the edge. Nevertheless, the worst case complexity of Algorithm 2 is also  $O(|E_l|)$ , where we usually have  $|E_l| < |E|$ .  $\square$

## 5. QUERY INDEX MAINTENANCE



The ultimate purpose of maintaining truss numbers for each edge is to serve queries. In the real world, a graph is often involving and thus the trusses are changing from time to time. At a particular time, if a user wants to know the current  $k$ -trusses, it is desired we can answer this query as soon as possible. In this section, we will introduce how maintaining truss numbers for each edge can help answer the queries quickly, and what other maintenance work we need to do, i.e. query index maintenance.

## 5.1 Querying the $k$ -trusses

To find the  $k$ -trusses from a graph, if we do not have the truss numbers recorded on the edges, we may need to recompute the  $k$ -trusses from scratch if the graph has been updated since the last query, because the  $k$ -trusses may change after the updates. Apparently, recomputing the  $k$ -trusses is very time-consuming, even though some pruning methods can be designed to speed up the process, e.g. pruning the edges that have global support less than  $k - 2$ . On the other hand, if we have the truss numbers recorded and maintained, things will be easier. A simple method is to traverse the graph and group connected edges with truss number  $k$  together. Each connected component is then a  $k$ -truss.

## 5.2 Indexing the $k$ -trusses

To avoid traversing the whole graph to answer a  $k$ -truss query, we can take one edge from a  $k$ -truss and represent the  $k$ -truss with the edge which we call a representative. In this way, the  $k$ -truss containing the representative can be found by performing a depth-first or breadth-first traversal from the representative on the graph. To be specific, let the representative be  $e$ , in the traversal, when we meet other edges with truss number  $\phi(e)$ , we include the new edges into the truss containing the representative, and finally we can find a truss with truss number  $\phi(e)$ . As a result, if we want to know all the  $k$ -trusses, we just need to find out  $k$  edges each of which is in a standalone truss, and then find the trusses by traversing the graph from those edges. That means if there are  $t$   $k$ -trusses, we can index  $t$  edges to represent the  $t$   $k$ -trusses.

How to choose a representative? In general, a representative can be chosen randomly. It takes similar cost to find a  $k$ -truss starting from different representatives within the  $k$ -truss.

## 5.3 Maintaining the index

In this section, we introduce how to maintain the index, i.e. maintain the representatives.

### 5.3.1 Deletion

Firstly, we discuss a deletion. If a deletion occurs, a number of edges will be affected from the deleted edge  $e$ . For a particular truss number  $k \in [k_{\min}(e), \phi(e)]$ , there are two types of change related to  $k$ -trusses: (1) some  $(k + 1)$ -truss edges become  $k$ -truss edges; (2) some  $k$ -truss edges become  $(k - 1)$ -truss edges. As to the representatives, there are three cases: (1) some  $k$ -truss representatives may stay as representatives because their truss numbers stay the same; (2) some  $k$ -truss representatives may stop being representatives because their truss numbers change to  $k - 1$ ; (3) in addition, we may need to add some new representatives into the index to represent some newly generated  $k - 1$ -trusses.

Consider the original  $k$ -truss  $T_k$  containing the deleted

edge  $e$ , after deleting  $e$ , we may further delete some other edges (starting from  $e$ ) whose truss numbers downgraded to  $k - 1$  in  $T_k$ . As a result, the updated  $T_k$  may shrink into a smaller  $k$ -truss, or may be decomposed into several  $k$ -trusses, or may even disappear ( $T_k$  becomes a part of a  $(k - 1)$ -truss). To maintain the representatives for the  $k$ -trusses, the method is to search the updated  $T_k$  and choose a representative for each connected component. If a representative is not in the current  $k$ -truss index, we add the new representative into the index. If a previous representative has the truss number downgraded from  $k$  to  $k - 1$ , then remove the representative from the  $k$ -truss index. In this way, truss index is maintained for deletion.

For each edge, we can record its representative. For the edges in the non-affected area, if their representatives are deleted, we then choose themselves as representatives, and propagate the information onwards until all the bridges between the non-affected area and the affected area have been examined. For the affected area, we explore from the deleted edge to find a connected affected area with truss number  $k - 1$  (the affected area had truss number  $k$  before the deletion). If this area is connected with an edge represented by an edge  $e_{rep}$  with unchanged truss number  $k - 1$ , then all the edges in the connected affected area will be represented by the representative  $e_{rep}$ , otherwise, we randomly choose an edge from the connected affected area as a new representative to represent the area.

### 5.3.2 Insertion

Secondly, we discuss an insertion. If a insertion occurs, similarly a number of edges will be affected from the inserted edge  $e = (a, b)$ . For a particular truss number  $k \in [k_{\min}(e), \min(|S_{a,b}| + 1, k_{\max}(e))]$ , there are also two types of change related to the  $k$ -truss: (1) some  $k$ -truss edges become  $(k + 1)$ -truss edges; (2) some  $(k - 1)$ -truss edges become  $k$ -truss edges. For the representatives, (1) some  $k$ -truss representatives may stay as representatives; (2) some existing  $k$ -truss representatives may be removed, because, after inserting the edge  $e$ , some existing  $k$ -trusses can be merged into a larger  $k$ -truss, so we only need one edge to represent the larger  $k$ -truss; finally, we may need to add some new  $(k + 1)$ -truss representatives if there are newly generated  $(k + 1)$ -truss.

Again, let  $T_k$  be a  $k$ -truss before inserting  $e$ , where  $k$  is in the range  $[k_{\min}(e), \min(|S_{a,b}| + 1, k_{\max}(e))]$ , (1) if  $k > \phi(e)$ , the  $k$ -truss representatives remain the same, because  $e$  is not in any  $k$ -truss, and thus does not affect the  $k$ -truss indexes; (2) if  $k \leq \phi(e)$ , we start from the edge  $e$  and traverse the subgraph  $T_k \cup \{e\}$  by inspecting the connected edges with truss number  $k$ . During the process, if we meet more than one existing representatives, we keep one of them in the index and delete other representatives from the index. If we do not meet any existing representative, we add one of the edges with truss number  $k$  into the index to represent the current  $k$ -truss. In this way, truss index is maintained for insertion.

Similarly, for a non-affected area, the truss numbers of the edges are not changed. The edges will be represented by the same edge representatives even though the edge representatives may upgrade from  $k$ -trusses to  $k + 1$ -trusses. This is because a  $(k + 1)$ -truss representative can also be a  $k$ -truss representative, since the representative can be guaranteed to be in a  $k$ -truss. There may be repeated representatives

**Table 2: Datasets**

	vertices	edges	maximum truss
Epinions network	75879	508837	33
Enron email network	36692	183831	22
Slashdot social network	77360	905468	34

representing the same connected truss subgraph. The repetition will be resolved when querying the  $k$ -trusses when repeated representatives will be removed. For the affected area, for  $k \in [k_{min}(e), \min(|S_{a,b}| + 1, k_{max}(e))]$ , we do the following: if an affected edge  $e'$  has the truss number increased to  $k + 1$ , but the truss number of its representative is still  $k$ , we change the truss representative of  $e'$  into  $e'$  itself, and then propagate the effect to the neighbor edges.

## 6. EXPERIMENTS

In this section, we report the performance of the truss maintenance algorithms. All experiments are done on a desktop with Intel(R) Core(TM) i5-2400 CPU at 3.1GHz and 8GB RAM. The operating system is Windows 7 Enterprise, and the code is written in Java.

### 6.1 Datasets and Approaches

We use three datasets to test the algorithms, Epinions social network (soc-Epinions1) [16], Enron email network (email-Enron) [17], and Slashdot social network (Slashdot0811) [17]. Epinions social network is a who-trust-whom online social network of a general consumer review site (www.Epinions.com). Members of the site decide whether to trust each other. Enron email communication network covers all the email communication within a dataset of around half million emails, which are collected and published by Federal Energy Regulatory Commission. Slashdot social network is a network that contains friend links between the users of Slashdot. All the above datasets are in Stanford Large Network Dataset Collection<sup>2</sup>. We give the details of each dataset in terms of number of vertices, edges, and average degrees in Table 2. We randomly generated enough number of updates (insertions and deletions) for each dataset and stored them for reuse. We guarantee that when we test different approaches on a particular dataset, we use the same set of updates. The approaches we have compared are listed in Table 3

### 6.2 Effect of batchUpdate vs progressiveUpdate

In this section, we compare the time cost of batchUpdate and progressiveUpdate. The batchUpdate approach stands for processing the updates in a batch and then doing truss decomposition on the graph to find the  $k$ -truss from the graph. The decomposition starts from smaller truss numbers and stops at the number  $k$ . The progressiveUpdate approach means that the algorithms introduced in Section 4 are used to maintain truss numbers while performing updates and the  $k$ -trusses are evaluated by searching the graph using the truss numbers. Fig 8 shows that progressiveUpdate is much better than batchUpdate when the number of updates is moderate for all the datasets. As the number

**Table 3: Summary of the approaches**

batchUpdate	process the updates in a batch mode and do a truss decomposition to find the new $k$ -truss
progressiveUpdate	maintain the truss number for each edge, process the updates on-the-fly and search the graph to find the new $k$ -truss using the recorded truss numbers
indexedUpdate	maintain the truss number for each edge, maintain truss representatives as an index, process the updates on-the-fly and find the new $k$ -truss by searching the graph from the representatives

of updates increase, the gap between batchUpdate and progressiveUpdate decreases. The cost of progressiveUpdate will catch up with batchUpdate when there are 10,000 updates. Fig. 9 and 10 have similar trends, but the catch-up points are different.

### 6.3 Effect of index vs non-index

In this section, we compare the performance of progressiveUpdate and indexedUpdate. Compared with the progressiveUpdate approach, indexedUpdate additionally builds and maintains the truss indexes, i.e. truss representatives. Queries will be answered by traversing the graph from the representatives and the representatives are maintained when processing edge updates. For most datasets, it can be seen that indexedUpdate is better than progressiveUpdate when the number of updates is moderate. As the number of updates becomes larger, indexedUpdate becomes worse than progressiveUpdate, such as the result shown in Fig. 8(d), 9(d) and 10(d). The reason is probably that indexedUpdates spends more time on maintaining the indexes.

### 6.4 Effect of different truss number $k$

In this section, we report the effect of different truss number  $k$ . As we can see, for all datasets, for larger numbers of  $k$ 's, progressiveUpdate and indexedUpdate are much better than batchUpdate, i.e. they can accommodate more updates before they become worse than batchUpdate. For example, for Enron email data, 8,000 updates make progressiveUpdate worse than batchUpdate for  $k=18$  (see Fig. 9(b)), while only 6,000 updates make progressiveUpdate worse (see Fig. 9(d)). The reason is that, for a larger  $k$ , only a small portion of the graph needs to be accessed when processing edge updates. A larger  $k$  implies a smaller  $k$ -truss. For smaller  $k$ 's, the advantage of progressiveUpdate and indexedUpdate compared to batchUpdate become less obvious.

### 6.5 Effect of different datasets

In this section, we report the effect of different datasets. In Fig. 10, for Slashdot data (average clustering coefficient: 0.0555), progressiveUpdate and indexedUpdate are better than batchUpdate up to 16,000 updates for  $k=30$ , while for Epinions data (average clustering coefficient: 0.1378), progressiveUpdate and indexedUpdate are only better than batchUpdate under around 10,000 updates (Fig. 8(a)). The

<sup>2</sup><http://snap.stanford.edu/data/>

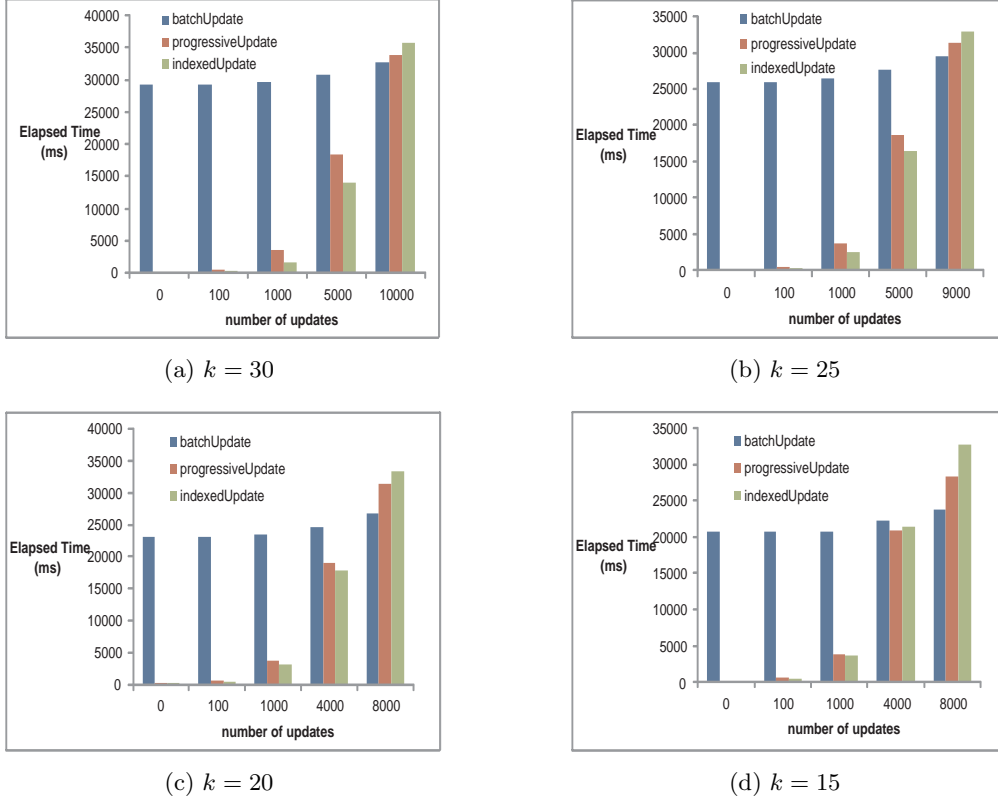


Figure 8: Epinions Social Network

reason is that Slashdot data is relatively sparser and is not a centralized dataset. As a result, the updates are usually done less costly. Epinions data is more centralized than Slashdot data, and thus each update needs to access more nodes. Among the three datasets, Enron email network (average clustering coefficient: 0.4970) is the smallest dataset, on which all the approaches run faster, however, progressiveUpdate and indexedUpdate allow the smallest number of updates before perform worse than the naive batchUpdate approach. Our maintenance approaches work better for decentralized data.

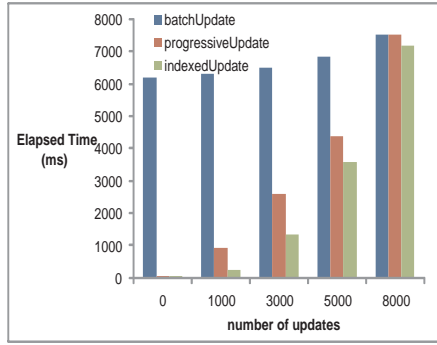
## 7. RELATED WORK

The concept of “truss” was firstly introduced by Cohen [1] in 2008, when social networks developed fast and corresponding research prevailed. Compared with other cohesive subgraph models, such as clique [5], quasi-clique [6, 7],  $n$ -clique [8],  $n$ -clan [9],  $n$ -club [9],  $k$ -plex [10],  $k$ -core [11],  $k$ -truss has its own advantage (mentioned in the introduction already). The computation of  $k$ -truss, also called truss decomposition, has been studied in [1, 18, 15]. Cohen proposed the first truss-decomposition algorithm [1], which is later outperformed by an improved in-memory algorithm proposed by Wang and Cheng [15]. Wang and Cheng proposed an out-of-memory algorithm for truss decomposition and a top- $t$   $k$ -truss evaluation algorithm [15]. Cohen [18] also proposed a map-reduce based algorithm for truss decomposition. Recently, Zhao and Tung [19] studied the truss decomposition problem and consider that the networked data is stored in a graph database. They also studied how to

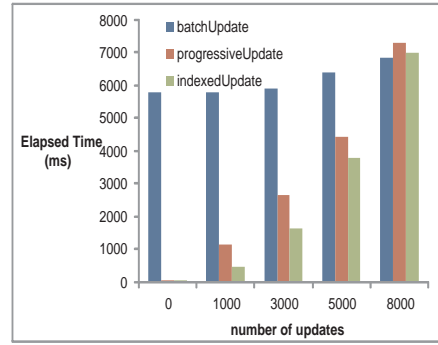
visualize the graph. Different from all the above works, we do not consider finding the trusses from scratch. We aim to maintain the trusses in face of frequent updates.

Another closely related work to ours is  $k$ -core maintenance [20]. Although the “maintenance” theme is shared in these two works, the technical solutions are different, because  $k$ -core and  $k$ -truss impose requirements on different aspects. To be specific,  $k$ -core imposes requirements on the nodes, asking each node to connect to at least  $k$  other nodes; while  $k$ -truss imposes requirements on the edges, asking each edge to be in at least  $k - 2$  triangles. Moreover, in our work, we also discussed how to build  $k$ -truss indexes and how to maintain  $k$ -truss indexes. These corresponding questions were not discussed for the  $k$ -core counterpart in the work [20].

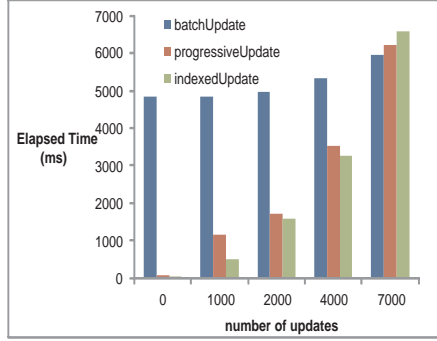
All the models above can be considered as explicit models for discovering cohesive structures from networked data. The common feature is that a structure with certain property is predefined, and then the rest work is to design efficient algorithms to discover all the subgraphs with the structure requirement. Another stream is called implicit models, some proposed objective functions first, such as modularity [21], normalized cut [22], and then partitioned the graph into a number of parts where the objective functions can be maximized or minimized; Some works defined neighbourhood distance, such as propinquity [23], structure closeness [24], and then grouped nearby nodes within a distance threshold around a given node to form a group; Some borrowed the idea of Markov Clustering [25, 26] to repeat *random walk* for a few rounds until self-organized clusters turn up. The



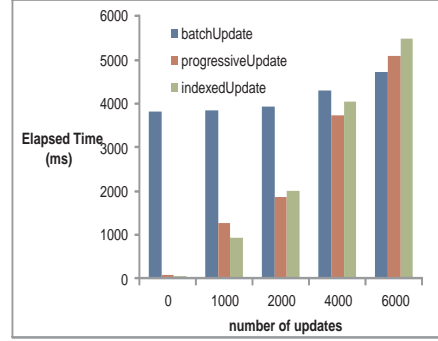
(a)  $k = 22$



(b)  $k = 18$



(c)  $k = 14$



(d)  $k = 10$

Figure 9: Enron Email Network

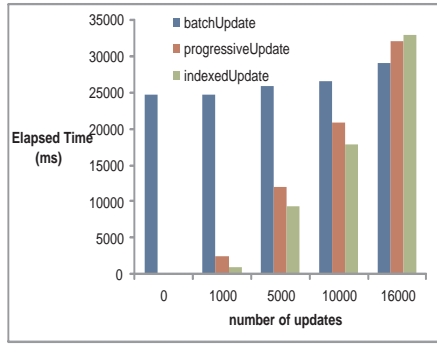
maintenance problem for implicit models are not studied as well.

## 8. CONCLUSIONS AND FUTURE WORK

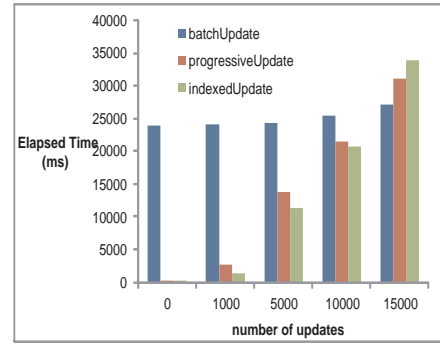
In this paper, we have studied how to maintain trusses in an evolving network by considering edge deletions and insertions. For one edge deletion or insertion, we have investigated the properties of truss change, proposed algorithms to perform truss maintenance and introduced techniques to maintain truss index updates. In the experiments, we have shown that maintaining truss on-the-fly is more efficient compared to performing batch edge updates and then doing truss decomposition if the update number is not large. We have also shown that having truss indexes and maintaining truss indexes are superior than evaluating truss queries without index support for moderate update frequency. As far as we can see, one direction for future work is to process data that cannot fit into memory, e.g., a  $k$ -truss may be very large when  $k$  is small.

## 9. REFERENCES

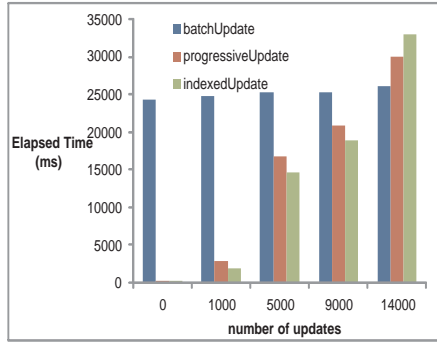
- [1] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. 2008.
- [2] Manfred Meyer, Markus Balsam, Arlo O’Keeffe, and Christian Schlüter. Admotional: Towards personalized online ads. *IJCSA*, 8(2):59–80, 2011.
- [3] Jacqueline Johnson Brown and Peter H Reingen. Social ties and word-of-mouth referral behavior. *Journal of Consumer Research: An Interdisciplinary Quarterly*, 14(3):350–62, December 1987.
- [4] Leena Rao. Linkedin now adding two new members every second. *TechCrouch.com*, Aug 2011.
- [5] R. Luce and Albert Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.
- [6] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. Massive quasi-clique detection. In *LATIN*, pages 598–612, 2002.
- [7] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM Trans. Database Syst.*, 32(2):13, 2007.
- [8] R. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15(2):169–190, 1950.
- [9] Robert Mokken. Cliques, clubs and clans. *Quality & Quantity: International Journal of Methodology*, 13(2):161–173, 1979.
- [10] Stephen B. Seidman. A graph-theoretic generalization of the clique concept. *Journal of Mathematical Sociology*, 6:139 – 154, 1978.
- [11] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269 – 287, 1983.
- [12] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. Finding maximal  $k$ -edge-connected subgraphs from a large graph. In *EDBT*, pages 480–491, 2012.
- [13] András A. Benczúr and David R. Karger. Randomized approximation schemes for cuts and flows in



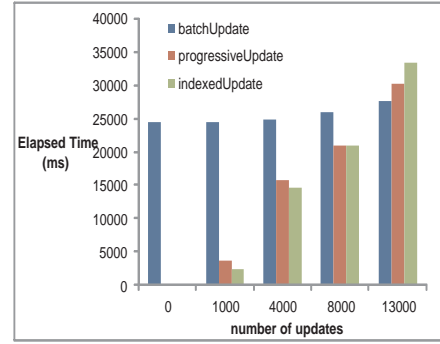
(a)  $k = 30$



(b)  $k = 25$



(c)  $k = 20$



(d)  $k = 15$

**Figure 10: Slashdot Social Network**

- capacitated graphs. *CoRR*, cs.DS/0207078, 2002.
- [14] Weifa Liang, Richard P. Brent, and Hong Shen. Fully dynamic maintenance of  $k$ -connectivity in parallel. *IEEE Trans. Parallel Distrib. Syst.*, 12(8):846–864, 2001.
- [15] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, May 2012.
- [16] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. Trust management for the semantic web. In *International Semantic Web Conference*, pages 351–368, 2003.
- [17] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.
- [18] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engg.*, 11(4):29–41, July 2009.
- [19] Feng Zhao and Anthony K. H. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. *PVLDB*, 6(2):85–96, 2012.
- [20] Rong-Hua Li and Jeffrey Xu Yu. Efficient core maintenance in large dynamic graphs. *CoRR*, abs/1207.4567, 2012.
- [21] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.
- [22] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, 2000.
- [23] Yuzhou Zhang, Jianyong Wang, Yi Wang, and Lizhu Zhou. Parallel community detection on large networks with propinquity dynamics. In *KDD*, pages 997–1006, 2009.
- [24] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. Scan: a structural clustering algorithm for networks. In *KDD*, pages 824–833, 2007.
- [25] Venu Satuluri and Srinivasan Parthasarathy. Scalable graph clustering using stochastic flows: applications to community discovery. In *KDD*, pages 737–746, 2009.
- [26] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. *J. Graph Algorithms Appl.*, 10(2):191–218, 2006.